Under Construction: Delphi And Strings

by Bob Swart

In this article, we'll examine the different kinds of strings that can be found in Delphi. From the good old short strings (which aren't so good anymore, as we shall see) to the long, ANSI and wide strings. From simple characters to pointer-to-characters, plus ANSI and wide characters. There's a lot going on behind the scenes to make it work just fine for the average Delphi programmer without having to know these nitty-gritty details.

Sometimes, however, it sure helps if you know just that little bit more about the implementation details of strings. This article will help you understand them. So, without further ado, let's start the first round with the building blocks of strings: characters.

Typical Characters

Delphi knows three character types: `Char`, `AnsiChar` and `WideChar`. There are actually only two different types: `AnsiChar` and `WideChar`. The default `Char` type is just an alias for the current recommended character type, which is `AnsiChar` at this time.

`AnsiChar` gets you an 8-bit ANSI character, which is indeed the default under Windows, at least on my installations of 32-bit Windows (Win95, Win98, WinNT and Windows 2000).

A `WideChar`, as you might expect, is twice as big (wide) compared to a regular character. In these 16 bits, we can fit a Unicode character. A Unicode character consists of two bytes. The first 256 Unicode characters map to the basic (ANSI) character set. Unfortunately, while Windows NT and Windows 2000 have support for Unicode, Windows 95 does not. I guess that's the reason why Delphi's support for Unicode (ie `WideChar` and, as we'll see in a moment, `WideString`) is a bit limited to say the least.

There are a number of special character functions that we should look at already (because I want to use some of them in a moment), namely `Ord`, `Chr` and `UpCase`. `Ord` returns the ordinal number of the character, while `Chr` turns an 8-bit byte value into a character again. Note that there is no support for `WideChar` here: you have to explicitly cast a 16-bit `ShortInt` to a `WideChar` in order to produce one. Similarly, the built-in `UpCase` function only works with ANSI characters, not with `WideChars`.

Null Termination

Apart from these single characters, Delphi also supports pointer-to-character types (with which you are probably well acquainted) such as `PChar`. And where `PChar` is a Pointer-to-`Char` (the 8-bit `AnsiChar` that is), you won't be surprised by now to learn that Delphi also supports the `PAnsiChar` and `PWideChar` types. All `PChar` types are in fact null-terminated strings, which means that they can be any length, but have to end with a character that has the hex value 0 (that is, null). `PChars` were actually introduced in Turbo Pascal for Windows: they were needed because almost the entire Windows API uses these null-terminated strings (when they need to pass a string, that is). Null-terminated strings are also the way in which you can pass a `PChar` string from Delphi (Pascal) to a `char*` string in C or C++.

The biggest problem (or just something not to forget) when using null-terminated strings is the fact that you need to allocate memory for them. An exception to the rule can be made when you initialise a `PChar` with a `const` string value (for which the compiler then allocates memory). Another way to avoid having to manually allocate and de-allocate memory is by declaring a zero-based array of character, which is type-compatible with `PChars` (if you compile with `$X+`, which is the default setting).

To give you an example, consider the statements in Listing 1. After the assignment `Y := X;` the `Y PChar` variable points to the `X` zero-based array of characters. And if we modify `Y` (or rather: the array of characters `Y` points to), we modify `X` as well. Not earth shattering, but just something to keep in mind, as it can easily lead to some confusion later in your code (been there, done that).

Short Strings

A legacy from the 16-bits Turbo Pascal days (even before null-terminated strings were supported), short strings are still quite popular among Delphi programmers. A short string is a string that can contain up to 255 characters. You can specify the maximum length when you declare the short string, for example:

```
var
  NameStr: String[64];
```

declares a variable `NameStr` which is of type `String` (or `ShortString`) with a maximum length of 64 characters. Each character is stored in a single byte of type `Char`, and you can access the individual characters using `NameStr[1]` for the first character to `NameStr[64]` for the last character. A reference to `NameStr[65]` will lead to a compiler error. Furthermore, if you compile using the `$R+` compiler directive,

➤ *Listing 1*

```
var
  X: Array[0..42] of Char;
  Y: PChar;
begin
  X := 'this is a null-terminated string';
  Y := X; { pointer Y points to X now }
  Y[0] := 'T';
  writeln(X);
```

```
function CaptionMaker1(Str: ShortString): ShortString;
{ stack: 256 (arg) + 256 (local) + 256 (result) bytes }
var
  i: Integer;
  Tmp: ShortString;
begin
  Tmp := Str; // initial copy
  for i:=1 to Length(Str) do
    if (i = 1) or (Str[i-1] = #32) then
      Tmp[i] := UpCase(Str[i]);
  Result := Tmp
end;
function CaptionMaker2(Str: ShortString): ShortString;
{ stack: 256 (arg) + 256 (result) bytes }
var i: Integer;
begin
  for i:=1 to Length(Str) do
    if (i = 1) or (Str[i-1] = #32) then
      Str[i] := UpCase(Str[i]);
  Result := Str
end;
function CaptionMaker3(const Str: ShortString): ShortString;
{ stack: 256 (result) bytes }
var i: Integer;
begin
  Result := Str; // initial copy
  for i:=1 to Length(Str) do
    if (i = 1) or (Str[i-1] = #32) then
      Result[i] := UpCase(Str[i])
end;
procedure CaptionMaker4(var Str: ShortString);
var i: Integer;
begin
  for i:=1 to Length(Str) do
    if (i = 1) or (Str[i-1] = #32) then
      Str[i] := UpCase(Str[i])
end;
function CaptionMaker5(var Str: ShortString): PShortString;
var i: Integer;
begin
  for i:=1 to Length(Str) do
    if (i = 1) or (Str[i-1] = #32) then
      Str[i] := UpCase(Str[i]);
  Result := @Str
end;
```

➤ *Listing 2: CaptionMakers.*

then a reference to `NameStr[i]`, with `i` of type `integer`, will lead to a runtime out-of-bounds error if `i` is not in the range 0 to 64. Yes, I did say 0, and the `NameStr` variable of type `String[64]` is actually 65 bytes long, since the 'zero-th' position of any short string contains the length of the actual string, stored in a `Byte` (which is the main reason a short string cannot contain more than 255 characters).

The main advantage of short strings over null-terminated strings is speed. It only takes one quick look (at the zero-th position) to determine the length of the short string, while you need to traverse the entire null-terminated string to determine its length.

However, short strings always had one big potential problem (especially in 16-bit days) and that was stack space. Even today, you should make sure you use short strings properly inside recursive routines, since they are placed on the stack, which is quite slow and has a finite limit, even in Win32.

Short CaptionMaker

As an example, consider a routine that will change the first letter of each word in a given sentence to uppercase. Let's call it `CaptionMaker`. The algorithm is simple: change the first letter to `UpCase`, and then change every letter that follows a space. A real version of this routine would of course check for additional characters, not just a space, such as tabs, carriage returns, linefeeds, etc.

Listing 2 has five possible implementations of the `CaptionMaker` routine. The first one takes a short string argument (256 bytes on the stack), returns a short string (another 256 bytes) and finally uses a local short string variable (the third 256 bytes allocated on the stack). The second one doesn't use a local short string variable, but modifies the passed string parameter in place, before returning that result. This saves 256 bytes on the stack compared to `CaptionMaker1`. The third edition passes the input string as a `const` parameter, which saves another 256 bytes on the stack.

`CaptionMaker4` does not even return a result, but changes the input string as a `var`-argument in place. This is the most efficient edition and uses the least stack space. If you still require a function, then `CaptionMaker5` shows how you can return a pointer-to-a-string as well.

Timings

Using the old technique of the RepsTimer (as described by TurboPower about a decade ago), I've executed all five versions of `CaptionMaker` during a 0.1 second period and counted the number of calls I could make in this time. A higher number indicates a faster performance. The results using Delphi 5.01 (showing values in K, or 1,000 repetitions) shouldn't be surprising: the algorithms using the least amount of stack space were most efficient; the procedure edition, using almost no stack space at all, was almost twice as fast as the first edition:

```
1: 52K
2: 65K
3: 74K
4: 95K (most efficient)
5: 83K
```

Results on your machine may vary. Note that for the best (undisturbed) results, you need to run on WindowsNT or Windows 2000, since Win95 and Win98 have problems shielding one application from another (which results in a lot of noise).

We'll come back to this example later when we use other kinds of strings.

Long Strings

A long string, which is also called `AnsiString`, is a welcome enhancement to the short strings that have been available for years. Instead of being limited to 255 characters, long strings can contain, in theory, up to 2 billion characters (actually 2Gb minus 1, but who's counting), in practice limited to the amount of free memory and swap file space on your machine. I've been playing with some long strings of several megabytes in size, but only for efficiency demonstration purposes. One of the best things about long strings is the fact that they can change in size: you can start with a relative small long string, and it will grow when needed. Note that this growing of long strings can take considerable time, as a new long string must be allocated, the old information copied, and so on. Listing 3 has an ultra-slow example to read a text file that will quickly (or slowly) demonstrate this.

Apart from going slower with every next line to read (a file with twice as many lines will take far more than twice the time to load this way), the above algorithm will also lead to a highly fragmented set of memory allocations for the long string, each one a bit bigger than the previous one. This is not a good thing, of course.

Reference Counted

Apart from the fact that long strings can grow in size (when needed or requested), an important feature of them is the fact that they are reference counted. This means that if you have two copies of the same long string, then you usually only have one actual copy with a reference counter set to 2. This sounds nice, and is indeed nice most of the time, but it can lead to some unexpected performance hits when working with long strings.

For example if you change a single character inside a long string variable, which was actually a copy of another long string (so they're both pointing to the same long string with a reference count value of 2), then you need to make a deep copy of the long string before you modify it. And that deep copy takes time: first allocating enough memory, then actually copying the string, and so on. In my Delphi Efficiency conference sessions (for example at the recent BorCon 2000 in San Diego) I always call this the 'delayed performance hit' of long strings.

When you think you are facing such a situation, then you can enforce each long string to be unique by calling the `UniqueString` procedure. This makes sure that the long string passed as an argument has a reference count of exactly one. Of course, when you do this your application may require more memory, but at least you won't be surprised by it.

Long CaptionMaker

Now, let's see how long strings perform when we modify the `CaptionMaker` routines to use `LongString` instead of `ShortStrings`. Since a long string is just a pointer, and even a long string value argument will just be a pointer that is being passed, we should not see the big differences that we have seen when using short strings.

```
procedure ReadFile(const FileName: ShortString): String;
var
  f: Text;
  Line,Str: String;
begin
  Str := '';
  Assign(f,FileName);
  Reset(f);
  while not eof(f) do begin
    readln(f,Line);
    Str := Str + #13#10 + Line { re-allocate Str when needed }
  end;
  Close(f);
  Result := Str
end;
```

➤ *Listing 3*

The results of our standard test are as follows, using Delphi 5.01:

```
1: 25K
2: 34K
3: 36K
4: 71K
5: 41K
```

Wow, compare that to the short strings, and you see that long strings are *much* slower. But *why?* Well, to tell you the truth, the loop that I've been using to measure the amount of repeats is programmed as shown in Listing 4.

While this is a perfectly working loop when testing short strings, we must remember that long strings are not only reference counted but also lifetime-managed. Which means that whenever a long string gets out of scope (or its lifetime is over by any other means), then the reference counter is decreased, and the long string itself is possibly destroyed (that is, the memory is deallocated again). And guess what happens when we assign the result of `CaptionMaker` to `Str2` again? The previous value gets lost, that's what. Meaning that the previous long string assigned to `Str2` must be removed, cleaned-up and then de-allocated. Which of course takes time. And hence the effect that long strings *appear* to be slower than short strings.

Is this surprising? Well, I'm rather glad if it is, because some day you will encounter such a case of unexpected performance hit in the real world.

```
Reps := 0;
EndTime := TimeGetTime + 100;
repeat
  Inc(Reps);
  Str2 := CaptionMaker1(Str);
until TimeGetTime > EndTime;
writeln('1: ',Reps);
```

➤ *Listing 4*

Delphi 4 Versus Delphi 5

And if you think the previous eye-opener was surprising, check this one. If you recompile the same application with Delphi 4 again and run it, you'll find that the results are always at least 10% better using Delphi 4. And no matter what or how you try, long strings in Delphi 4 are faster than in Delphi 5. The results are as follows:

```
1: 30K
2: 40K
3: 43K
4: 77K
5: 43K
```

What's the explanation? Well, it turns out that the reference counter of long strings wasn't really completely thread-safe in Delphi 4 and earlier. It was only thread-safe when writing, but not when reading. To fix that, from Delphi 5 on, all access to the reference counter of a long string requires an actual `LOCK` statement. The effect may vary from machine to machine, but at least speed won't go up (although thread-safety will), that's for sure!

Further Delphi 5 Issues

What is even more worrying is that, according to several sources, short strings are turned into long strings (behind the scenes) in Delphi 5 prior to many string operations. This affects the efficiency of older applications, and is

something that can best be seen if you recompile your application using Delphi 4 and Delphi 5 and compare the timings.

Note that at BorCon 2000 in San Diego it was made clear that short strings will not be removed from Delphi. In fact, we've even heard that they will still be in Kylix. Regardless of that, I have no idea what the reasoning is behind this conversion of short strings to long strings. It's not always done and the `CaptionMaker` didn't suffer from it, but when in doubt, just compare the Delphi 4 and Delphi 5 timings to be sure.

HyperString

If you're really concerned with the speed of `AniStrings`, then you should consider working with HyperString, a freeware library by EFD Systems, see http://efd.home.mindspring.com/hyperstr.htm for more details. HyperString contains a few hundred routines to work with `AnsiStrings`. There are freeware DCUs for Delphi 3, Delphi 4 and Delphi 5, and you can buy a source code license for $59 if you want. I can recommend this library if you're concerned about string performance.

Wide Strings

A wide string is a long string that is made of special characters: `WideChars`. While all the strings we have covered so far were made up of single byte characters, Windows has some great support for multi-byte characters, which was not found in Delphi until the support for so-called wide strings became available.

First a warning: `WideStrings` are based on the Unicode implementation of your operating system (which isn't supported by Win95 for example). As a result, some of the code that will follow might not work on your machine exactly the way it works on mine. It's the idea that counts, of course.

Compared to `LongStrings`, `WideStrings` have some serious drawbacks. Apart from the fact that they are twice as big (every character is a `WideChar`), they are also slower, because `WideStrings` are not reference counted. How would that affect the `CaptionMaker` routine? Well, the results are as follows (they are virtually the same using Delphi 4 or 5, by the way):

```
1: 20K
2: 24K
3: 33K
4: 73K
5: 45K
```

`WideStrings` indeed seem to be slower than long strings. They still need to be cleaned up, and are a bit bigger, so there is more to be cleaned up at that.

Note that for `WideString` compatibility (or rather, `WideChar` verus `Char` incompatibility) I even had to rewrite the loop to make sure that we checked the `Ord` of each character against 32, and assign the `WideChar` value of `UpCase` of the `Chr` of `Ord` of the `WideChar` to the `WideChar` again. Oh well, here's the code (like I said earlier, it would be nice if Delphi had some more `WideString` and `WideChar` support inside):

```
for i:=1 to Length(Str) do
  if (i = 1) or
    (Ord(Str[i-1]) = 32) then
    Str[i] := WideChar(UpCase(
      Chr(Ord(Str[i]))));
```

Delphi does contain three functions that support conversion of null-terminated `WideChar` strings to regular `AnsiString` or `WideString` variables. These are `WideCharToString`, `WideCharLenToString` and `StringToWideChar`.

WideString And COM

Would we need `WideStrings` in Delphi even if we never (think) we need Unicode support? Well, if you want to use COM as a way of application communication, then you should know that COM-based strings are, indeed, `WideStrings`. Or `BSTR` as they say in the COM world.

An `AnsiString` may even be converted behind the scenes into a `WideString` for you, which is even less efficient that using a `WideString` directly from the start.

ResourceStrings

Yet another string type is called the `ResourceString`. This is only a type definition to help you declare string constants that will be placed in a resource file (for easy translation or localisation of your application). As with any string constant, resource strings are in fact short strings and limited to 255 characters in length.

And in case you missed the implication: any constant string is in fact a short string expression. So the definition

```
const
  Message = 'Hello, world!';
```

declares a string constant, that can only be changed by recompiling the application, whereas

```
resourcestring
  Message = 'Hello, world!';
```

declares a resource string that ends up in the string table linked with your executable (which you

Using Unicode

Windows NT and Windows 2000 support diverse languages and character sets using Unicode, a 16-bit character set that unifies Asian, Arabic, and many other character sets in a single standard. Delphi has some support for Unicode, but the VCL must maintain compatibility with Windows 95, which does not support Unicode. In fact, supporting Unicode controls seems to be an area that Borland overlooked entirely.

If you must use Unicode controls in your application, you need to write your own. Delphi gives you full access to the Windows API, but it lacks design-time support for `WideStrings`. The Object Inspector, for example, does not understand Unicode. The component streaming system does not support Unicode, either. Ray Lischner is preparing an article for the October 2000 issue that will show you how to overcome these limitations and use Unicode strings at design-time.

can edit and modify by any self-respecting 32-bit Windows resource editor such as Resource Workshop).

What, Where And When?

So, which string type should you use, where and when and why? By default, I would use normal `Chars` (8-bit `AnsiChar`) and normal `Strings` (the long string type, also known as `AnsiString`).

When using old-style record types that contain string fields, you have no choice but to use `ShortStrings`. But, apart from that, I would seldom fall back to short strings these days (especially since Delphi 5 will make sure they'll often be turned into long strings before working with them anyway).

Whenever I define some string constants that may need to be translated, I go for the `ResourceString` keyword. This is much easier than modifying the strings and recompiling the application (it even prevents the burden of having to maintain an external windows string resource file, since it just creates one for you behind the scenes).

Most, but not all, of the functions that make up the Windows API use C-type null-terminated strings, so there's `PChar` for that. And for plain communication with other environments (without using COM, that is) you should also rely on simple `PChars`.

COM uses a type called `BSTR` which is a Unicode wide string, compatible with the `WideString` type of Delphi. So, when using COM and interfaces (where you expect to communicate with other environments) you may wish to investigate `WideStrings` instead.

Final Access Violation

Long and wide strings are like dynamic arrays: Delphi will check the index (subscript) and, if it's out of range, you'll get a range check error (when compiling with `$R+`, that is). However, an empty long or wide string is represented internally by a nil pointer, and that will always result in an access violation.

Bob Swart (aka Dr.Bob, www.drbob42.com) is an @ Consultant for TAS Advanced Technologies, freelance technical author and Delphi Trainer who has spoken at the Inprise/Borland Conferences since 1993.